Spring 3-1-1984

Towards an Intergrated Environment for Accessing External Databases ; CU-CS-263-84

Dennis M. Heimbigner
University of Colorado Boulder

TOWARDS AN INTEGRATED ENVIRONMENT FOR ACCESSING EXTERNAL DATABASES

by

Dennis Heimbigner*

CU-CS-263-84 March, 1984

* University of Colorado at Boulder, Department of Computer Science, Campus Box 430, Boulder, Colorado, USA.

Towards An Integrated Environment for Accessing External Databases

Dennis Heimbigner

Department of Computer Science
University of Colorado
Boulder, Colorado 80309

ABSTRACT

Diverse is a system that is designed to provide sophisticated access to external database systems. The requirements for such a system are outlined and the Diverse architecture is proposed for meeting those requirements. The Diverse system is built upon a relational database system for storing data and a knowledge base for interpreting procedural information. Diverse is intended to be an integrated environment consisting of a directory of information about the operation and semantics of external databases and media, a syntax-directed database translator, and a dialogue system for mechanizing routine operations.

Introduction

Accessing and integrating information from various sources is a fundamental problem in any office situation. These sources consist of the various databases available throughout a company, as well as the externally available commercial systems such as Dialog, Compuserve, and Dow-Jones.

At the moment, such databases can be accessed in one of two ways. First, the databases may actually be elements of a common distributed database system. In this case, a user may only have to learn one language, and, usually, there are mechanisms for combining data from the various databases. This may be fine for small, homogeneous, systems, but it is unlikely to encompass the variety of available data sources.

A second way to access external data is to simulate a terminal calling into the system. While it does provide access, it provides no help in understanding and structuring the data obtained in this way. Each of the databases has a different command structure and a different model for structuring data. Thus, a user must know or learn many different languages in order to access various systems. Even when you have the data, it may be difficult to combine it with data from other sources. Some of it may be in database format and some in text format. Furthermore, it may be difficult to extract the relevant parts from the mass of data that you have collected.

It is reasonable to search for some middle ground: a system that is capable of more sophisticated access than terminal simulation, but that does not require total integration before useful data sharing can be achieved. The goal of this paper is to outline the requirements for such a system, and propose an architecture, termed Diverse, that is capable of achieving these goals.

Requirements for Diverse

The principle characteristics of external data sources are autonomy and heterogeneity. Such systems often are commercial ventures over which a user has no control. They are free to maintain their data in any way they wish, and they are not required to provide special interfaces to various users of their system. As a consequence, the user is forced to provide his own capabilities for integration, local to the workstation, and independent of the external data provider.

Since these systems are autonomous, it is unlikely that they will all adopt identical, or even similar, interfaces and data structures. The user is forced to deal with the variety of languages provided by these external databases. Of course, there is always some pressure to standardize, but historical inertia can often prevail. For example, all of the systems mentioned above (Compuserve, etc.) have quite different command languages. In fact, some of them have several different languages depending upon which database is accessed.

Even if a common interface is provided, the user will still see only a collection of independent data sources. Mechanisms must be provided for transferring information between the various sources and for identifying inter-database relationships.

Additionally, there must be provision for guidance and abstraction for the various activities involved in accessing data. It must provide step-by-step sequencing of these activities, it must provide knowledge about the external system, and it must allow the user to extend his knowledge to newly available databases.

Finally, there must be guidance in dealing with the broad spectrum of communications media available now or in the foreseeable future. These would include telephone, cable, local area networks (e. g., Ethernet) and satellites. Each

medium has protocols that must be observed, and our systems must understand those protocols and diagnose the possible problems involved in actual use.

Related Work

This research draws from a number of previous projects and disciplines. The architecture presented here comes most directly from the work on federated databases [Heimbigner 82]. Briefly, the federated database architecture is an attempt to provide maximal autonomy for existing database systems, yet still provide substantial sharing of data. It is characterized by the lack of a global schema and its reliance on a distributed dialogue system. to coordinate the activities of the component databases.

It is clear in retrospect that the federated architecture has its place, but it relies heavily on having the same software at all sources of data. System R* [Lindsay 80] is an another system emphasizing autonomy, and it also suffers from the same problem. In the current environment where there are many independent databases, and none of them use the same software, neither System R* nor federated databases is appropriate.

The RITA project (Rand Intelligent Terminal Access) [Waterman 78] is also similar to Diverse. RITA was an attempt to provide an intelligent agent for programmers. It was based upon a production system that attempted to learn from the activities of the programmer. It also could deal with external systems, such as the Arpanet.

Diverse is specifically set up to handle external database systems, while RITA focuses on the act of programming. In theory, RITA could provide smart-terminal access to external databases, but it does not appear that it could provide much help in interpreting and integrating data obtained from those external sources. In addition, RITA spends much of its effort in automatic learning: a very difficult problem in artificial intelligence. Diverse has no such capabilities; rather, explicit user actions are used to enlarge its information base.

Lochovsky and Tsichritzis at the University of Toronto are also exploring the problem of accessing external databases [Lochovsky 81]. Their emphasis is on the user interface, and it implicitly uses a form of videotex (page-oriented) model. Under this framework, all data is cast into pages of data. While the user interface is important, The Diverse system attempts to provide more sophisticated understanding of the contents and structure of the external databases. It too casts much of that data into a fixed form (the relational model), but its use of text as common model may allow other structures to be used. Finally, the Toronto work does not yet promote integration of data from several sources.

An Architecture for Diverse

Diverse may be divided into four components: the execution system, the simulation system, the translator, and the user interface. The execution system provides support for the data and procedures defined in the other activities. The simulation system must maintain state information about the currently active databases and communications media. It uses the execution system to set up the simulation and to interpret it. The translator is responsible for converting external database output into local database format. The user interface is intended to provide a clean presentation of the actions of the simulation.

The Execution System

The execution system provides reasonably general long-term storage of information about external databases and a programming system to execute programs. Corresponding to these two divisions, it contains a database and a knowledge base.

The data for the system is stored locally in a relational database system. The database actually contains two connected collections of data. Most of the data is stored in a file system on secondary memory. This acts as a long-term repository for data for other components. But, the normal database is augmented with additional data that is short-term and kept in primary memory. This latter database is used by other components to keep highly structured, but volatile, state information. In some sense, then, the volatile database is used for its structuring capabilities as opposed to its storage capabilities.

The relations in Diverse are somewhat unorthodox because the attribute values may contain both data values and procedures. When one want to connect to a particular system, the tuple for that system is extracted from the database and inserted into the environment. This insertion will set appropriate data values and also insert certain procedures (such as logon, top-level, extract, and display) into the knowledge base.

In addition to the database, the execution system contains a knowledge base. It contains procedural knowledge for other components, and it provides an interpreter for that knowledge. For several reasons, Diverse uses a variant of Lisp as the representation language:

1. The interpreter is relatively small.
2. The language is simple.
3. The language is flexible enough to support the programming techniques used in Diverse.

The Lisp system is augmented with facilities for using knowledge about external databases. In particular, it supports a simple, Prolog-style, system of rules. Such rule systems have been in common use in artificial intelligence research for a number of years, and the basic principles are well understood.

A rule system consists of a database of information and a set of rules acting upon that database. In Diverse, the database is an in-core relational database provided by the database agent of Diverse. A rule consists of a left hand pattern that is satisfied when all of the right hand elements have been satisfied. For example, one (slightly simplistic) rule might be:

```
connect(Database) :- dial(Database), logon(Database).
```

This says that the connect clause is satisfied when the dial clause is satisfied and the logon clause is satisfied.

Simulation

Diverse attempts to be knowledgeable about external systems by maintaining an internal "simulation" of the possible states of the external system and also maintaining. procedures for driving the system from state to state. States are defined in user terms, and are defined in terms of the input modes and accessible external objects. Thus, to obtain some piece of financial information about a company under the Dow-Jones database, the external system must be in a state

that allows one to ask the appropriate question. If the external system is not in the right state, it must be driven to that state from the current state.

In addition to the simulation of the states, Diverse must provide procedures whose action depends upon the current state of the system. Thus, to disconnect from some systems, one must get to the "top level" and then type a specific command such as logoff. Performing the disconnect operation in some particular state first requires getting to the top level, which in turn may depend upon the current state.

The term simulation may be a bit misleading, but it is intended to refer to an internal representation of both the state of the external database system and the state of the medium connecting Diverse to the database. The simulation creates an environment with a number of data items and operations whose semantics are determined by the medium, the external database, and the current state of that database.

Consider again the example for reaching a specific external database: the connect operation, defined generically as follows:

```
connect(Database) :-    get_dir_info(Database),
                        dial(Database),
                        logon(Database).
```

This says that to connect to a database, one must get the appropriate directory information, dial the database over some medium, and then perform the logon sequence for that database. The first operation searches the directory database for the specified database name. If it finds it, it inserts data and procedures into the environment. The next two operations (dial and logon) in turn search the environment to find some defined method for dialing and logging onto the specified database. In fact, these definitions will have been set up by the get_dir_info operation and so they will succeed (assuming that get_dir_info succeeded).

The directory mentioned above represents information about known external systems. It maintains both data and knowledge about these external systems. For example, it may know the appropriate media and phone numbers for accessing some database; this data would be inserted into the environment to define concrete values for use by procedures that actually establish the connection. Other data might consist of user-ids and passwords, and access sequences for reaching specific databases that are contained in a more global system (e.g. specific databases under Compuserve).

The directory also contains procedural knowledge (rule-sets) that uses the data to achieve certain ends. Thus it may have knowledge about the procedure for login and logout, and the procedure for taking access sequences and actually reaching a specific database. In addition, it may contain meta-knowledge about the system. Such meta-knowledge may describe the means for learning about the existence and structure of new databases.

The Diverse system can expect to access external databases through many different communications media, possibly simultaneously when data from several sources must be correlated. In order to accomplish this, Diverse must store knowledge about the various media. Note that the media data must be separate from the directory because there is not a one to one correspondence between communications media and databases. Certainly the phone system allows one to access many different external systems, for example.

The media database consists of information about each medium: transmission rate, parity, cost per minute, and so on. In addition to raw data, the media

database must provide some procedural knowledge about the medium. Thus, it must have some idea about how to connect to a medium, disconnect from it, and how to recognize transmission errors when they occur.

Dialogue

The ultimate purpose of Diverse is to allow a user to perform some activity. This activity may involve simply reading mail, or finding the latest quote for a stock, or it may involve sophisticated inter-database manipulations. Many of these activities can be routinely performed by Diverse once they have been set up by the user. The dialogue subsystem provides partial automation of many of the routine steps of these various procedures. A dialogue system specifies the structure and semantics of these activities by specifying the sequence of steps as well as the data involved in each step.

A dialogue system is effectively just another programming language. It is distinguished from normal programming languages by the fact that it is provides specialized operations. This means that it has some very powerful primitives for doing some domain dependent activities. The dialogue language for Diverse must provide operations specific to accessing external database: operations such as connect, enterdatabase, query, display, and disconnect.

The dialogue language must have two special features: exception handling and concurrency. Failures and errors are inherent problems in accessing external computers. The external system may fail or the connecting medium may fail or introduce errors. Certainly anyone who has had to use a noisy phone line knows how difficult it is to do useful work. Exception handling in most languages is extraordinarily difficult. Those that can (e. g., Ada) are too complicated to serve as a dialogue language. When exceptions are combined with concurrency, the difficulties are compounded.

Most dialogue systems cannot handle errors in a concise fashion, and unfortunately, Diverse is no exception. At the moment, errors occurring during dialogues must be explicitly tested. Ornamenting the dialogue with explicit tests seriously complicates the structure and makes it hard to understand and verify. As a last resort, if the error is sufficiently catastrophic, the dialogue is terminated.

The other feature that is needed is concurrent execution of dialogues. A user may set up some procedures that must run automatically:

(1) They may periodically gather information commonly referenced by a user. Examples are the latest stock quotations, or department progress reports, or electronic mail.

(2) They may periodically update local information. For example, a user may be interested in a particular class of information that is kept locally. An automatic dialogue may then periodically scan the external databases for new information, and if it is found, it may bring it to the attention of the user.

(3) They may periodically perform information searches. The user may have arbitrary information searches that he wants to perform at off hours when it is less expensive. He may construct a dialogue for performing that search automatically.

Concurrent dialogues are essentially independent processes devoted to the execution of the body of some dialogue. The dialogues must have some means of synchronizing themselves with the state of the database so that they can recognize the occurrence of important events. Diverse uses a restricted predicate for

this purpose. As an example, suppose that a user had a dialogue called "stories" that would collect all of the stories about a given company. Furthermore, suppose that the user wanted to watch the price of the set of companies in his portfolio and when the price dropped by 10%, he wanted to find the recent articles on that company. We might construct the following dialogue:

```
stockcheck() :-

        IF stockquote(Stock,Start,Hi,Lo),
           (Lo - Hi) > (0.1 * Start)
        THEN
           stories(Stock).
```

The process is started when stockcheck is invoked. The IF THEN clause will be instantiated for every stock that satisfies the condition. If one occurs, then the body of the clause is executed and, in this case, stories is invoked to get the stories on the company.

This predicate mechanism is intentionally similar to the production systems such as OPS5 [Forgy 81]. As in OPS5, the predicates that Diverse can handle are restricted to those that depend upon the existence of tuples in database that meet certain simple criteria. Currently, whenever a new tuple is added to a database, predicates over that database are evaluated. The intent in restricting the form of predicates is to eventually allow for optimization of the predicate evaluations.

Translator

Up to now, we have assumed that (somehow) the output of an external database can be placed in the local database in a relational form. The translator is the system responsible for this transformation. The operation of the translator is to take a text representation of data from some other system and translate it into the relational model for direct manipulation. The translator takes both text and an annotated grammar as input. The grammar is used to parse the input and to specify the translation into relations. This is accomplished by annotating the parts of the grammar that correspond to the fields and specifying how to reassemble those fields into tuples in a relation. The appropriate grammar depends upon the current database and the current state of the database. For example, The Dialog database system has two forms of output: one for search results and and another for display results. The search result indicates how many documents satisfied some condition. After that, one may ask for the display of the documents in the set. Interpreting each of these results requires a very different grammar. Whenever Diverse changes state it must ensure that the appropriate grammars are inserted into the current environment.

Many external databases assume that the user is viewing data on a screen terminal. Often, the clues to the extent and meaning of the output are determined visually: that is, by the placement of the data on the screen. It is relatively easy for a person looking at the screen to understand the format, but it is more difficult for a context-free parser. As a result, the grammar for the screen must include certain kinds of spacing information so that a correct parse is produced. Traditional parsers for programming languages tend to ignore spatial boundaries because they are either unimportant to the language, or they have been removed by ad-hoc lexical fixes. For example, most Fortran compilers take this latter approach. Tagging an LALR grammar with spatial marks does not extend the power of the grammar, but it is important to provide a natural method for

specifying the tags and their location.

User Interface

The final element of the Diverse system is the user interface. It must provide the user with a reasonably consistent view of the system and allow him to perform his job. Since a number of activities may be performed concurrently, the interface must provide for separate interface to each. In addition, a user may want to vary the level of detail, and so the interface must be able to expand and contract the amount of information that it displays.

It would appear that some form of windowing system [Teitelman 77] is an appropriate basis for the user interface. On top of this layer, the simulation must have some graphic or linguistic representation that the user can manipulate to achieve his desired ends. It is clear that an effective user interface is essential to the use of Diverse, but it must be one of the last pieces to be completed since it depends upon the language forms provided by the other agents.

Examples

The Knowledge Index by Dialog Information Services is a keyword in text system that maintains many different kinds of databases: financial, journal references, and computer software listings, for example. In order to extract information from that system one must perform two steps: search and display. Dialog allows a user to specify a boolean expression involving search terms. Dialog then shows you the number of matching terms. For example, asking "FIND XYZ CORP? AND ANNUAL REPORT" would return:

```
        100   XYZ CORP?
      11133   ANNUAL REPORT
S1        6   XYZ CORP? AND ANNUAL REPORT
```

The last line indicates that Dialog has created a set, S1, of references matching the search. Given this, one may display elements of this set in varying modes. Typing "DISPLAY S1/L/1" causes a long listing of the first element of S1, which might be the actual annual report of XYZ.

In trying to access this external database, Diverse must recognize its structure and then act on the information embedded in it. Thus it must break up the search answer and place it into a relation, called DIALOG_SETS, as follows:

DIALOG_SETS	set	size	term
	1	6	"XYZ CORP? AND ANNUAL REPORT"

Given this relation, Diverse must decide how to continue. In particular, it may decide to pull all of the elements in the set into its database, or it may decide that the set size is to large. If it is too large then it must come up with a strategy to reduce the size. In order to do this, the the knowledge base is invoked. Some of the rules in the KB would be:

```
display(Set,Size,Def) :-
                    (Size < 5),
```

```
                    dialog_extract(Set,Size,grammar).

display(Set,Size,Def) :-
                    (Size >= 5),
                    restrict(Set,Size,Def,Newset,Newsize,Newdef),
                    display(Newset,Newsize,Newdef).

display(Set,Size,Def) :-
                    (Size >= 5),
                    dialog_extract(Set,5,grammar).
```

The first rule says that if the size of the set is less than 5, then invoke dialog_extract to print out the set elements, parse them using the appropriate grammar, and insert them into an appropriate database. The second rule says that if the set size is 5 or more, then try to invoke a routine to restrict the old set and to create a new set that is smaller than the given set. This restriction might attempt to add extra terms to the definition based upon synonyms for terms. If the restrictor succeeds, then attempt to display the new set instead of the old one. The last rule, which would be invoked if all else fails, extracts only the first 5 elements of the set and ignores the rest.

It may sometimes be necessary to use information from one database as input to another, which is, in effect, an attempt to link information between disjoint databases. Consider the "stories" dialogue mentioned above. It will look for recent news stories on a Corporation so that the user can peruse the stories. Note that no attempt is to be made to interpret the meaning of these stories, only to search for them. The stock price information may have been obtained from Dow-Jones, but corporate news is available both from Dow-Jones and Dialog, and it is expected that both will be searched. Additionally, duplicate stories from both sources should be purged.

To perform the stories operation, the stock name must be obtained, and then connected back to the name (or names) of the company. Sometimes a subsidiary of the company will cause perturbations in the parents stock and so the company must be connected to its parent or its subsidiaries. The final result must be a set of terms of interest.

Once the names are obtained, it is time to search for stories about them. First, the Dow-Jones system is invoked via connect. Next the appropriate database is entered to look for headlines about the companies. For Dow Jones, this is essentially a loop to ask for stories about each term in turn. Each story is collected and inserted into a stories database using the company as the key. Once the Dow-Jones stories are collected, it must be disconnected and Dialog connected in its place. As mentioned before, this will put Dialog specific operations into the environment for later use by this dialogue. Now, the term set is used to extract stories from Dialog and insert into the database.

Once the stories are collected, they must be filtered to purge those that are duplicates or are not of recent origin. Age may be determined if the story has an associated date. Duplicate may be eliminated by searching for common keys for the stories. For stories, a common key would be the magazine, the date, the title, and the pages. For each duplicate, one of the two entries is purged.

The final example is a grammar for Dialog search results. It would be invoked as

```
extract(DIALOG_SETS, search_grammar).
```

It is assumed that whenever Diverse sends a command to the external database,

that the result is a new screen of information to be treated as a result. For sufficiently smart databases, this may require Diverse to simulate a set of screen operations to keep an accurate map of the state of the screen as it might be seen by a user. Dialog, though, can be treated in this simple minded manner because it does not do any significant screen manipulation. Thus, the extract command applies the grammar "search_grammar" to the current screen and places the resulting tuples into the relation DIALOG_SETS.

A partial grammar for search_grammar would be as follows:

```
search      : setmatch
            | match search
            ;

match       : LEFTBORDER integer predicate RIGHTBORDER ;

setmatch    : LEFTBORDER set integer/2 predicate/3 RIGHTBORDER ;

set         : "S" integer/1 ;
```

A search output is defined to be a series of zero or more match lines followed by one match line with an attached set name. The grammar is in a slightly modified Yacc notation. Each rule is a non-terminal, a colon, and a sequence of left sides separated by vertical bars. Terminals are in upper-case. Note the use of LEFTBORDER and RIGHTBORDER to match the beginning and end of screen lines. It is expected that when this grammar is used, it will extract three fields: the set number, the count, and the predicate. All three will be extracted as strings, assembled into a tuple, and inserted into the relation DIALOG_SETS. The fields are marked into the grammar using the notation "/1", "/2", and "/3". When this grammar is applied to the screen, certain parts match the nodes marked as fields. The actual field value is the substring that corresponds to the terminals of the subtree rooted at the field node. For this example, those substrings corresponds to two integers and a Dialog format predicate (grammar not given).

Control Issues in Diverse

For completeness, the Diverse architecture must eventually address a number of issues of control: protection, concurrency, and recovery. Because the external databases are autonomous and heterogeneous, mechanisms traditionally used for distributed databases are not applicable here.

Data protection is at best limited when accessing external databases. The best that one can hope for is that the external databases provide some level of protection for their data. It is impossible for an external database to protect the data once it is sent out over some communications medium. Thus, someone using Diverse will be free to capture information from some external database. But once that data has been captured, there is no way to prevent the user from sharing that information with whom ever he pleases.

Concurrency appears at two levels in the Diverse system. First, there is concurrent access to the database and knowledge base components within Diverse. It seems that traditional database approaches (transactions plus some form of locking) will be adequate to handle this situation.

The second level of concurrency occurs when a user wishes to read and write data from external databases. We cannot use locking and so we may find

ourselves using data that is changing during the period of time it is being accessed. Often this will not pose a problem because the changes are infrequent, or because the user doesn't care if there is some minor inconsistency.

Although locking is not possible, it may be that a modified optimistic concurrency control [Kung 81] can be used when a high degree of consistency is required. For instance, the transaction might be executed and then a second transaction executed that checked the results of the first one. If the check is successful, then it is assumed that the original transaction was correct. If the check fails then the original transaction may have operated on inconsistent data, and recovery must be invoked.

Most recovery in database systems uses some form of commit protocol with delayed write operations. Such an approach is not viable in Diverse because there is usually no way to delay the writes on autonomous systems. Again we must propose a less than perfect solution: reversible transactions [Gray 81]. For every transaction provided by an external database, we must specify another transaction that can, for practical purposes, reverse the effect of that transaction. Obviously there are irreversible transactions, but in many cases, there may be approximate reversals that suffice.

Interaction with Diverse

Now and in the near future, Diverse will not be simple to use. Constructing dialogues and grammars requires considerable programming skill. An eventual goal is to allow non-programmers to effectively use the system.

One solution may be to use examples and scripts of actual interactions. When a user first contacts a new database, he knows very little about it; he may have an user id and password plus a users manual in paper form. If Diverse were sufficiently "intelligent" if might be able to learn on its own by watching the user interact with the database. This is a serious artificial intelligence question and one to be avoided here. Rather, Diverse will make it easy for the user to capture histories of interactions and later use them as templates for defining common activities and formats with respect to the new database.

Setting up the translator poses some particular problems because it requires the construction of grammars for various inputs and outputs. Typical grammar formalisms (Backus-Naur, for example) take time to construct and are somewhat difficult to understand. Diverse will attempt to use examples of the actual output as the basis for construction of the grammar. The user can display the data and mark it with relevant fields and relationships. Once that is done, the user can then graphically specify the transformations of the fields into database relations. Using this information, the translator can apply the rules to new data automatically.

In sum, the emphasis in Diverse is not on automatic learning, but rather on providing powerful tools that allow a user to easily specify the appropriate data structures and knowledge for a new database.

Conclusion

In this paper we have outlined the requirements for a system to support sophisticated access to external databases. We believe that the architecture outlined here, called Diverse, is capable of meeting those requirements. The Diverse system is built upon a relational database system for storing data, and a knowledge base of rules for storing procedural information. Diverse is an integrated environment supporting a simulation of the external database and the

connecting medium. It allows a user to perform high level operations and to set up event driven activities.

References

[Forgy 81]
Forgy, C. L.. "The OPS5 User's Manual", Technical Report CMU-CS-81-135, 1981. Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA.

[Gray 81]
Gray, J., "The Transaction Concept: Virtues and Limitations", *Proceeding of the Seventh International Conference on Very Large Databases*, pages 144-154, Cannes, France, 9-11 September 1981.

[Heimbigner 82]
Heimbigner, D. M., *A Federated Architecture for Database Systems*, Ph. D. Thesis, University of Southern California, also available as Technical Report TR-114, August 1982, Computer Science Department, University of Southern California.

[Kung 81]
Kung, H. T. and Robinson, J. T., "On Optimistic Methods for Concurrency Control", *ACM Transactions on Database* Systems 6(2):213-226 (June 1981).

[Lindsay 80]
Lindsay, B., and Selinger, P. G., "Site Autonomy Issues in R*: A Distributed Database Management System", IBM Research Report RJ2927, 15 September 1980, IBM Research laboratory, San Jose, CA.

[Lochovsky 81]
Lochovsky, F. H., and Tsichritzis, D. C., "Interactive Query Languages for External Data Bases", Computer Systems Research Group, University of Toronto, March 1981.

[Teitelman 77]
Teitelman, W., "A Display Oriented Programmer's Assistant", *Proceedings of the 5th International Conference on Artificial Intelligence*, Volume 2, Cambridge, MA, 22-25 August 1977, pages 905-915.

[Waterman 78]
Waterman, D. A., "Exemplary Programming in RITA", *Pattern-Directed Inference Systems*, pages 261-279, D. A. Waterman and F. Hayes-Roth (eds.), Academic Press 1978.